Characterization of data movement requirements for sparse matrix computations on GPUs

Süreyya Emre Kurt, Vineeth Thumma, Changwan Hong, Aravind Sukumaran-Rajam, P. Sadayappan
Department of Computer Science and Engineering
The Ohio State University
Columbus, Ohio, USA
{*kurt.29, thumma.6, hong.589, sukumaranrajam.1, sadayappan.1*}*@osu.edu*

***Abstract*—Tight data movement lower bounds are known for dense matrix-vector multiplication and dense matrix-matrix multiplication and practical implementations exist on GPUs that achieve performance quite close to the roofline bounds based on operational intensity. For large dense matrices, matrix-vector multiplication is bandwidth-limited and its performance is significantly lower than matrix-matrix multiplication. However, in contrast, the performance of sparse matrix-matrix multiplication (SpGEMM) is generally much lower than that of sparse matrix-vector multiplication (SpMV).**

In this paper, we use a combination of lower-bounds and upper-bounds analysis of data movement requirements, as well as hardware counter based measurements to gain insights into the performance limitations of existing implementations for SpGEMM on GPUs. The analysis enables the development of an adaptive work distribution strategy among threads and results in the highest performing SpGEMM code for GPUs.

***Keywords*-data-movement bounds, sparse matrix-vector multiplication (SpMV), sparse matrix-matrix multiplication (SpGEMM), graph analytics, hypergraph partitioning, GPU computing**

I. Introduction

Sparse matrix computations are at the core of many compute-intensive applications, both in scientific/engineering modeling/simulation as well as large-scale data analytics. A large number of graph algorithms can also be formulated in the language of sparse linear algebra on semi-rings. The GraphBLAS consortium is defining a sparse linear algebra API intended for such use in developing portable implementations of graph algorithms using efficient implementations of key sparse matrix operations.

The wide ranging uses for sparse matrix operations has resulted in significant recent interest in developing efficient GPU implementations for sparse matrix-vector (MV) multiplication [18], [7], [15], [17], [20] and sparse general matrix-matrix (MM) multiplication [14], [16], [1], [9], [5], [7], [13]. We note that several variants of sparse matrix operations are used, depending on which operands are sparse/dense. In this paper we only consider the most widely used variants for sparse MV and MM: i) sparse-matrix times dense-vector (SpMV), and ii) sparse-matrix times sparse-matrix (SpGEMM).

A surprising fact is that the performance (in GFLOPs) of efficient GPU SpMV implementations [15], [17], [20] is much higher than that achieved by the best current GPU SpGEMM implementations [13]. This is in stark contrast to the dense case, where the performance of matrix-matrix multiplication (a BLAS3 operation) is typically orders of magnitude higher than matrix-vector multiplication (a BLAS2 operation). A primary goal of this work is to attempt to gain insights into why this is the case. **Is the low performance of SpGEMM because of some inherent fundamental bottleneck, such as data movement requirements? If not, is there scope for significant performance improvement?**

We begin (**Sec. II-B**) by first documenting performance of dense/sparse MV/MM on an Nvidia Kepler K20c GPU, using a variety of sparse matrices from the Suite Sparse collection [6], [21] drawn from different application domains. We confirm that SpGEMM performance is consistently much lower than SpMV performance.

Focusing on data movement requirements for SpMV and SpGEMM computations, we seek lower bounds as well as upper bounds based on hypergraph partitioning (**Sec. III**). We explicitly enumerate the complete set of arithmetic operations needed to perform sparse matrix-matrix multiplication as vertices of a hypergraph, with each data element being modeled as a hyperedge incident on all arithmetic operations (hypergraph vertices) that use that data element. By performing hypergraph partitioning, we determine upper bounds for data movement for SpGEMM computation. We find that upper bounds are quite close to lower bounds, while actual data movement in SpGEMM implementations is orders of magnitude higher.

A challenge to gleaning insights into performance bottlenecks for SpGEMM is the fact that data elements from three sparse matrices are used in each elementary operation. Evan if two of the matrices are kept the same to perform C = A*A, different rows of A (with different sparsity patterns) are involved. In order to better control the variability and gain insights, we devised a set of experiments to perform SpGEMM on banded matrices, but represented in the CSR representation used in all SpGEMM implementations (**Sec. V**). Further, each banded matrix was also randomized via a random symmetric permutation of rows/columns.

The experiments with synthetic banded matrices provide useful insights that insufficient concurrency is likely a factor contributing to lowered performance. Using this insight, we devise an adaptive work distribution strategy using virtual warps (**Sec. VI**) for SpGEMM. Experimental results demonstrate that the new SpGEMM implementation achieves better performance than existing publicly available GPU SpGEMM codes, including Nvidia's cuSPARSE [18], [7], bhSPARSE [14], [16], and HybridSparse [13].

The paper makes the following contributions:

- It undertakes the first systematic exploration, to our knowledge, of the data movement requirements for general sparse matrix-matrix multiplication, using a

range of matrices drawn from different application domains..

- It uses hypergraph partitioning on explicitly enumerated graphs of the computational operations and data dependences in multiplying various sparse matrices, it is demonstrated that there is not some inherent lower bound on data movement that forces SpGEMM to incur much higher data movement than SpMV.
- It uses experimentation with synthetic banded matrices to gain insights into SpGEMM using the scatter-vector approach, and diagnoses inadequate thread-level concurrency as a likely cause of performance loss.
- It devises an adaptive work distribution strategy among threads for a scatter-vector based GPU SpGEMM implementation, resulting in the highest performing SpGEMM code for GPUs.

II. Background: Sparse MV/MM on GPUs

In this section, we provide background information on challenges in achieving high performance sparse matrix-matrix (SpGEMM) multiplication on GPUs.

A. Dense versus sparse MV

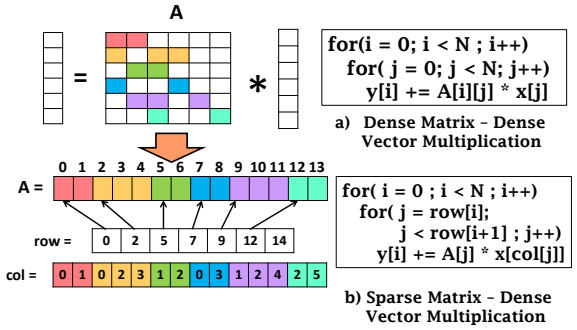

Figure 1. Dense Matrix Vector vs Sparse Matrix Vector Multiplication

Fig. 1 shows code for dense MV (top) and sparse MV (bottom) multiplication. The overall loop structure is very similar, with an outer loop running over rows of the matrix, and the inner loop performing a dot product of the vector with one row of the matrix. The main difference is in the representation and access of the rows of the matrix. The figure also shows an example of a sparse matrix in the CSR (Compressed Sparse Row) representation. The non-zero elements are grouped by row and compacted into a long 1-D vector (A) and a parallel 1-D vector (col) holds the column position of the corresponding data element. A third vector (row) holds indexes that point to the start of elements corresponding to each row. In the SpMV code, for processing row i, the inner loop runs over contiguously located data elements in A, from index row[i] to index row[i+1]-1. The needed element of the vector is obtained indirectly using col. Thus, the overall code structure is not very different between dense and sparse MV, with about double the data volume being needed in the sparse case because both the column index and the nonzero element value must be read in from memory; in contrast with dense MV, only the matrix elements need to be read in.

Algorithm 1: Dense Matrix-Matrix Multiplication

```
input : DenseMatrix A[M][N], DenseMatrix B[N][P]
output: DenseMatrix C[M][P]
1 for i = 0 to M-1 do
2     for j = 0 to P-1 do
3         C[i][j] = 0
4         for k = 0 to N-1 do
5             C[i][j] += A[i][k] * B[k][j]
```

B. Dense versus sparse MM

Next let us consider dense MM. Simple code for it is shown in Alg. 1 with three nested loops. Two of the three loop indices directly index the row/column of each of the three matrices. Each of the N^2 result elements C_{ij} is computed by accumulating N partial products $A_{ik} * B_{kj}$, for k ranging from 0 to N-1. This code does not show aspects like multi-level tiling necessary to achieve high performance, but illustrates a key characteristic that enables high performance: it is feasible to efficiently index and access the elements of the three matrices due to the simple direct relationship between indices of interacting data elements. Alg. 2 shows

Algorithm 2: Sparse-Matrix Sparse-Matrix Multiplication

```
input : SparseMatrix A[M][N], SparseMatrix B[N][P]
output: SparseMatrix C[M][P]
1 for each A[i][*] in matrix A do
2     for each non-zero entry A[i][j] in A[i][*] do
3         for each non-zero entry B[j][k] in B[j][*] do
4             value = A[i][j] * B[i][j]
5             if C[i][j] ∉ C[i][*] then
6                 Insert C[i][j] in C[i][*]
7                 C[i][j] = value
8             else
9                 C[i][j] += value
```

high-level pseudocode for SpGEMM. If the sparse matrices are represented in CSR format, efficient contiguous access to elements in any row is possible, but access to the elements in a column is not efficient. In order to compute the elements of a row i of C, all nonzero elements $A_{i,*}$ must be accessed, and for each such nonzero A_{ik}, all elements $B_{k,*}$ need to be accessed. For each such nonzero element B_{kj}, the product A_{ik}*B_{kj} must be computed and it contributes to a nonzero element C_{ij}.

A significant challenge in computing the sparse matrix product is in efficiently gathering together the various additive contributions to an element C_{ij} from different rows of B. Several approaches have been used to address this "index-matching" problem. One option is to use a hash table to store non-zero elements of C, and the implementation in the Nvidia cuSPARSE library [18], [7] and a recent implementation by Anh et al. [1] use this approach. An alternate approach has been to implement efficient "row-merge" functionality to merge contributions from two sparse

rows. This approach has been used by the implementation of Gremse et al. [9] and the bhSPARSE code from Liu and Vinter [14], [16]. To form row i of the result matrix C, the nonzeros A_{ik} first form vectors of partial products by multiplying with the nonzeros in B_{kj}, and then these sparse vectors of partial products are merged to form the result row $C_{i,*}$. Yet another approach, labeled ESC (Expand-Sort-Compress), was first developed for the Nvidia CUSP library [5], [4]. With this approach, all nonzero partial products A_{ik}*B_{kj} are first formed in parallel and written out as key-value pairs (i,j,value). The huge vector of key-value pairs is then sorted so that key-value pairs with the same row/column indices become adjacent. This is followed by a segmented prefix-sum computation to "compress" the key-value pairs by accumulating the values corresponding to the same row/column index. Finally, yet another approach is to use a dense "scatter-vector" of size N to solve the "index-matching" problem. This approach was first proposed by Gustavson [10] for the sequential context. Our prior work in developing the HybridSparse [13] SpGEMM implementation used a combination of the scatter-vector approach and the ESC approach. The scatter-vector approach is a focus also in this paper, and is described in greater detail later on in Sec. VI. zRectionPerformance of Dense vs. Sparse MV and MM

In this section, we present experimental data on achieved performance with state-of-the-art GPU implementations of dense/sparse matrix-vector (MV) and matrix-matrix (MM) multiplication. The surprising observation is the consistent reversal in relative performance of the two operations when considering the dense versus the sparse case: For dense matrices, performance of MM is much higher than MV, while the opposite is true for the sparse case.

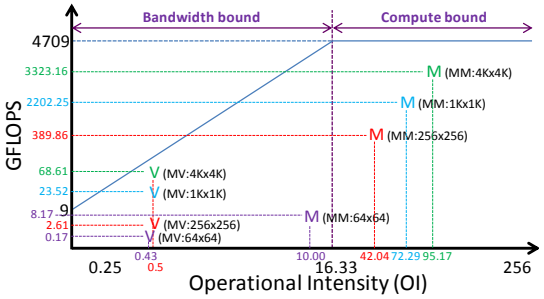

Figure 2. Roofline plot: Dense MV vs Dense MM

C. Performance of Dense MV vs. MM

Fig. 2 shows a roofline plot [22] for an Nvidia Kepler K20c GPU, with data-points for single-precision dense matrix-vector and dense matrix-matrix multiplication using Nvidia's cuBLAS library. A roofline plot provides insightful visual illustration of the extent to which algorithms are constrained by the data-movement bandwidth limits of a system. It contains two asymptotic lines that represent upper bounds on performance: i) the maximum computational rate of processor cores and, ii) the bandwidth from main memory to cores. The horizontal line represents peak computational performance (in GFLOPS), and the inclined line has a slope corresponding to the memory bandwidth (in Gbytes/sec). The y-axis of the roofline plot represents performance (in GFLOPs), while the x-axis represents the "operational intensity" (*OI*) of a computation, defined as the ratio of number of computational operations performed per byte of data moved between main memory and the processor cores. A code will be memory-bandwidth limited unless *OI* is sufficiently high, greater than a "critical intensity" corresponding to the point of intersection of the two rooflines. This is because the product of the *OI* and the peak memory-bandwidth (slope of the inclined roofline) imposes an upper-bound on the number of computational operations that can be performed per second due to the amount of data moved from/to main-memory.

Hardware counters were profiled using the Nvprof profiler for execution of dense MV and MM computations with cuBLAS library calls. For each problem size, the achieved performance in GFLOPS and *OI* (ratio of number of floating-point operations and measured data volume) were computed. The displayed points on the roofline plot show two clusters, one for the MV instances and another for the MM instances. For MV, since two floating point operations are performed per matrix element, the upper-bound on *OI* is 2 operations for 4 bytes transferred for single-precision computation, i.e., 0.5. The achieved *OI* is 0.43, very close to the theoretical limit.

Two clear conclusions that can be drawn from the roofline plot:

- Dense MV is bandwidth-bound, with achieved performance being quite close to the asymptotic sloping roofline representing the memory-bandwidth-based performance limit.
- Dense MM achieves over 30x performance than dense MV for large matrices, and the computation is clearly compute-bound – for large enough problem sizes, the plotted points are far to the right of the balance point where the rooflines intersect.

D. SpMV vs. SpGEMM Performance

We next present similar roofline data for sparse matrix-vector (SpMV) and sparse matrix-matrix (SpGEMM) multiplication using a collection of 25 sparse matrices from the Suite Sparse Matrix Collection [6], [21]. These matrices are drawn from a range of application domains and have been used in recent publications on optimizing SpGEMM [14], [16], [1], [13]. Characteristics of these matrices are provided in Table I. For SpMV, a state-of-the-art implementation using the CSR5 variant [15] of the compressed sparse row (CSR) data structure was used. For SpGEMM, we used the HybridSparse [13] code was used – it has been shown to achieve higher performance than any other publicly available GPU SpGEMM code. The matrix product C = A*A was performed using each of the 25 test matrices.

Since plotting the achieved performance at the measured *OI* for these 25 matrices on a roofline plot like Fig. 2 would make it very cluttered, we instead present the same information in a different form in Fig. 3. In the upper portion of the figure, the achieved performance for SpMV and SpGEMM are shown for all the sparse matrices. In contrast to the dense case, it may be seen that for every matrix the performance (in GFLOPS) achieved by SpMV is higher than that achieved by SpGEMM, with SpMV performance often being over an order of magnitude higher. The lower portion

Table I
CHARACTERISTICS OF SPARSE MATRICES: C=A*A

Matrix Name	NZ_A (K)	NZ_C (K)	#ops (M)	*OI* _UB
2cubes_sphere	1647	8975	55	2.237
cage12	2033	15232	69	1.794
cant	4007	17440	539	10.587
cit-HepPh	422	3713	13	1.468
com-amazon.ungraph	926	2531	6	0.816
com-dblp.ungraph	1050	4909	14	1.066
com-youtube.ungraph	2988	154931	373	1.166
cop20k_A	2624	18705	160	3.335
email-Enron	368	30492	103	1.649
facebook_combined	88	338	5	5.277
filter3D	2707	20162	172	3.361
m133-b3	801	3183	6	0.670
mac_econ_fwd500	1273	6705	15	0.817
majorbasis	1750	8243	38	1.633
mario002	2101	6450	26	1.204
mc2depi	2100	5246	17	0.888
offshore	4243	23356	143	2.241
patents_main	561	2281	5	0.813
poisson3Da	353	2958	24	3.213
rma10	2374	7901	313	12.371
scircuit	959	5223	17	1.215
web-BerkStan	7601	78351	445	1.213
web-Google	5105	29710	121	2.404
web-NotreDame	1497	16801	129	1.563
webbase-1M	3106	51112	139	3.311

of Fig. 3 displays the achieved *OI* by SpMV and SpGEMM for each matrix. Again, the trend is the opposite of that seen for the dense case: the achieved *OI* is always significantly higher for SpMV than than SpGEMM. This indicates that many more words of data are moved between main-memory and the cores for each FLOP with the SpGEMM code than the SpMV code. This is so although the ratio of operations to the data footprint is larger for SpGEMM than SpMV, as seen in Table I.

The following conclusions can be drawn regarding SpMV versus SpGEMM:

- In contrast to the dense case, performance of SpGEMM is considerably lower than performance of SpMV across all tested sparse matrices.
- The measured *OI* for SpGEMM is much lower than the measured *OI* for SpMV, which already is more memory bandwidth bound than dense MV. This is very different from the relative operational intensities achieved by dense MM versus MV.

The above observations raise the following important question: *Is SpGEMM inherently much more constrained than SpMV due to fundamental data movement requirements, or is it the case that even the best existing SpGEMM implementations on GPUs are very far from optimal in terms of data movement?* We begin to address this question in the next section.

III. DATA MOVEMENT BOUNDS FOR SPMV AND SPGEMM

For any computation, such as dense matrix-matrix multiplication of a given pair of input matrices, there are many

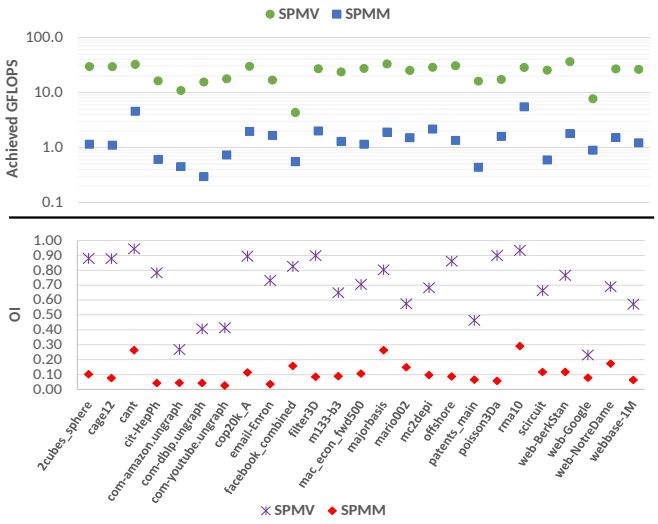

Figure 3. SpMM vs SpMV: Performance and Operational Intensity

valid schedules for the elementary arithmetic operations that collectively achieve the matrix multiplication. All equivalent schedules perform exactly the same number of arithmetic operations, but can differ very significantly in the number of cache misses incurred. Cache misses cause data to be moved across the levels of the cache/memory hierarchy. For very large problems that cannot fit even in the last level cache (LLC) in the memory hierarchy, the overheads for data movement between main memory and the last-level cache tend to be the most significant for sparse matrix computations. We therefore focus our attention on the data movement between a GPUs global memory and the L2 cache, which is the last-level cache in current Nvidia GPUs.

A. Data Movement Lower Bounds for SpMV and SpGEMM

Consider the execution of an algorithm for matrix-matrix multiplication of a pair of sparse matrices A and B to produce a resulting matrix C. The execution can be viewed in terms of a collection of elementary floating-point additions and multiplications. There will generally be many valid execution schedules corresponding to temporal reordering of the elementary operations. Each non-zero element C_{ij} of the result matrix requires additive accumulation of a number of contributions of the form $A_{ik} \times B_{kj}$ for matching non-zero elements in A and B. These operations can be interleaved in many ways to produce the same final result. All such valid schedules require exactly the same number of arithmetic operations, but can differ greatly in the number of data movements between the large but slow main memory and the fast but small cache/scratchpad store. It is of great interest to efficiently find valid schedules that minimize the total movement of data between the slow large memory and fast small cache. However, this is in general an open and unsolved problem for most algorithms.

Techniques have been developed to find data movement lower bounds for dense matrix-matrix multiplication [8], which generalizes to sparse matrix-matrix multiplication: $1.72 \times N_{Ops}/\sqrt{C}$, where N_{Ops} is the number of update (multiply-add) operations executed. However, this bound

turns out to be extremely loose compared to a simple data-footprint lower bound - total number of words needed to represent the input and output matrices. Since a CSR representation of a sparse matrix uses two words per nonzero (one for the column index and another for the value of the nonzero element), a lower bound on data volume for $C = A \times B$ is $2 * (nnz_A + nnz_B + nnz_C)$.

For SpMV, the data footprint is $2 * nnz$ for the sparse matrix, and $2 * N$ for the two vectors . This data footprint is a lower bound on the minimum possible data movement between main-memory and cache for execution of SpMV.

B. Upper Bounds for SpGEMM

In this section we describe our approach for determining a data movement upper bound for matrix multiplication $C = A \times B$, for specific matrices A and B. The main idea is to "tile" an explicitly enumerated large graph with one vertex for each elementary multiply-add operation needed for the matrix multiplication, and a hyperedge corresponding to each data element in A, B, and C, incident upon all operations using that data element. Loop tiling (also called loop blocking) is a well known optimization for nested loop computations, which changes the order of execution of the instances of the statements from the natural order specified by the iterators of the nested loops to a different one that groups contiguous blocks of statement instances in the multidimensional iteration space. The rationale for tiling is that often statement instances close to each other in the multidimensional iteration space exhibit data reuse. By performing the reordering to execute multidimensional blocks together, data reuse in a limited cache/scratchpad store is enhanced.

In the case of sparse matrices, simply tiling the 3D iteration space for sparse matrix-matrix multiplication with uniform sized tiles is not expected to be effective since different tiles will have highly varied operation counts. Instead, a hypergraph is formed that captures data reuse characteristics and operation reordering is achieved by performing hypergraph partitioning [3]. A hypergraph partitioner such as PaToH [3] places each graph vertex into a partition, with roughly equal number of vertices in each partition and a minimization of the number of inter-partition hyperedges. Thus, wherever possible a hyperedge is fully "internalized" to avoid an edge cut by placing all operations involving the corresponding data element within a single vertex-partition. The hypergraph partition is used to reorder the operations for sparse matrix multiplication to be executed in partition order: first execute all operations in the first partition, followed by all operation in the second partition, and so on. All multiply-add operations for the entire sparse matrix multiplication can be arbitrarily reordered. This is because i) the result elements in C can be formed completely independently of each other, allowing arbitrary interleaving in the multiply-add operations for different C_{ij}, and ii) associativity of addition means that the various multiply-add operations for a specific C_{ij} can also be arbitrarily reordered. Hence, the reordered execution after hypergraph partitioning is a valid execution order for the matrix multiplication.

Ballard et al. [2] used a form of "optimal" hypergraph partitioning of a graph comprised of the elementary arithmetic operations for sparse matrix-matrix multiplication to model data movement lower bounds. In this paper, we use a similar hypergraph partitioning approach, but we use it instead to form data movement upper bounds.

We first define the hypergraph for sparse matrix-matrix multiplication $C = A \times B$. The hypergraph construction is a modified version of the hypergraph construction of Ballard et al. [2]. Let A and B be $I \times K$ and $K \times J$ matrices, respectively. Let S_A and S_B be the sets containing the data elements of matrices A and B. The data elements of S_C are defined as the set containing elements:

$$S_C = \{(i,j) : (i,k) \in S_A \wedge (k,j) \in S_B)\}$$

Vertices $\mathcal{V}$ and Nets $\mathcal{N}$ of the hypergraph are defined as follows:

$$\mathcal{V} = \{v_{ikj} : (i,k) \in S_A \wedge (k,j) \in S_B\}$$

$$\mathcal{N} = \mathcal{N}^A \cup \mathcal{N}^B \cup \mathcal{N}^C$$

$$\mathcal{N}^A = \{n^A_{ik} : (i,k) \in S_A\} \quad n^A_{ik} = \{v_{ikj} : (k,j) \in S_B\}$$

$$\mathcal{N}^B = \{n^B_{kj} : (k,j) \in S_B\} \quad n^B_{kj} = \{v_{ikj} : (i,k) \in S_A\}$$

$$\mathcal{N}^C = \{n^C_{ij} : (i,j) \in S_C\} \; n^C_{ij} = \{v_{ikj} : (i,k) \in S_A \wedge (k,j) \in S_B\}$$

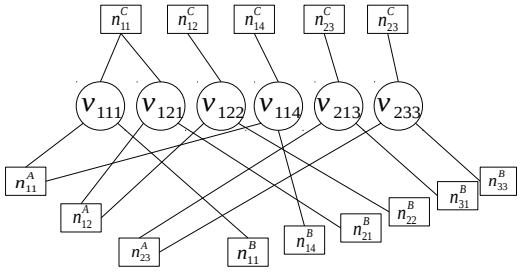

Figure 4. Hypergraph $\mathcal{H}(A,B)$ for $C = A \times B$ where S_A = {(1,1),(1,2),(2,3)} and S_B={(1,1),(1,4),(2,1),(2,2),(3,1),(3,3)}

Figure 4 shows an example of a hypergraph for SpGEMM. Each vertex in the hypergraph represents a multiply-add operation in $C = A \times B$ and each net represents a data element, either in A, B or C, and connects together all elementary operations that use that data element.

A key parameter to a hypergraph partitioner is the number of partitions. Our goal is to perform a partition such that the number of data elements touched i a partition (the number of incident hyperedges) does not exceed the cache capacity. We use an iterative strategy of performing repeated hypergraph partitions with increasing partition counts until this condition is satisfied.

A data-movement upper bound is computed by adding necessary store and load operations for each partition. With a CSR representation, for each element of A or B, we need to load 2 elements for it (1 load for column index and 1 load for element value). Similarly, for each element of C we associate 3 memory transactions for it (1 load for column index, 1 load for previous element value and 1 store for write-out of final value.

Alg 3 presents the algorithm for upper-bound computation. It partitions the hypergraph $\mathcal{H}(A,B)$ into $\mathcal{P}$ partitions. The minimum possible number of partitions satisfying the cache-size limit for their data footprint is unknown. So we

start with $P = 2$ partitions, and successively double the number of partitions until the cache capacity constraint is met. The loop at line 8 runs through each of the partitions to check for violations of the cache-capacity constraint. The loops at line 9, 12, and 15 accumulate data-footprint counts cuts for each partition, for array A, B, and C, respectively. Two words are accounted for each accessed data element (column index and element value).

Algorithm 3: Find Upper Bound SpGEMM

input : Hypergraph $\mathcal{H}(A, B)$, size of fast memory M
output: Data movement upper bound UB

1 $\mathcal{V} = \mathcal{H}(A, B).getVertices()$
2 $numPar = 2$
3 $pass$ = false
4 **while** $\neg pass$ **do**
5 $pass$ = true
6 $W_a[*] = W_b[*] = W_c[*] = 0$
7 par = Partition($\mathcal{H}(A, B)$,$numPar$)
8 **for** $0 < p \leq numPar$ **do**
9 **for** $(i, k) \in S_A$ **do**
10 **if** $\exists j, par(v_{ikj}) = p$ **then**
11 $W_a[p] = W_a[p] + 2$
12 **for** $(k, j) \in S_B$ **do**
13 **if** $\exists i, par(v_{ikj}) = p$ **then**
14 $W_b[p] = W_b[p] + 2$
15 **for** $(i, j) \in S_C$ **do**
16 **if** $\exists k, par(v_{ikj}) = p$ **then**
17 $W_c[p] = W_c[p] + 2$
18 **if** $\exists p, W_a[p] + W_b[p] + W_c[p] > M$ **then**
19 $pass$ = false
20 $numPar = 2 \times numPar$
21 **for** $0 < i \leq numPar$ **do**
22 $UB = UB + 2 \times W_a[i] + 2 \times W_b[i] + 3 \times W_c[i]$
23 $UB = UB - |S_C|$

A similar hypergraph partitioning was also performed for the SpMV computation to determine data movement upper bounds.

Fig. 5 presents data-movement upper-bound (UB) data from hypergraph partitioning, for SpGEMM and SpMV. Instead of absolute data-movement volume, ratios with respect to lower-bounds on data volume are presented. In addition to the UB/LB ratio, the ratio of actual measured data volume (presented previously in Sec. II-B in the form of operatonal intensity) to LB.

For SpMV (lower portion of Fig. 5), the UB/LB ratio is very close to 1.0 for all matrices. The measured-volume/LB ratio is also close to 1 for the majority of matrices, indicating that close to the minimal possible data movement is being achieved by the CSR5 SpMV implementation. For a few matrices drawn from social-network domains (e.g., com.amazon-ungraph and web-Google), the measured/LB ratio is very high. This is likely due to multiple uncoalesced accessed to the input vector elements during SpMV.

For SpGEMM too, surprisingly the UB/LB ratio is very close to 1.0 for the majority of matrices. This suggests that

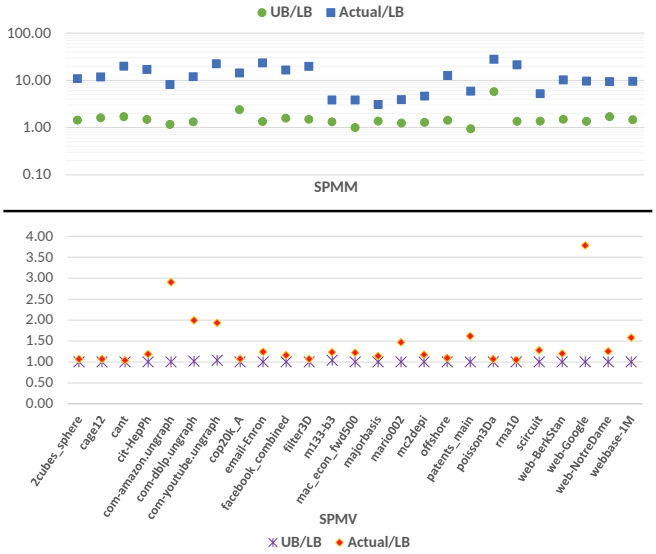

Figure 5. Ratios of Upper Bound and Measured Data Movement to Lower Bound for SpGEMM and SpMV

the capacity of L2 cache (around 1.5 Mbytes) in the Kepler GPU is sufficient to almost fully reuse data elements in cache, if a suitable reordering of the operations is performed. However, as observed previously, the ratio of measured/LB data volume is very high for most matrices for SpGEMM.

IV. Reordering Matrices

We can see from Figure 5 that the data-movement upper-bounds are significantly lower than the measured data movement for SpGEMM. This suggests that a reordered schedule of execution of the elementary operations (operations within a partition performed contiguously) has the potential to greatly reduce the total data movement between main-memory and cache. However, this would require the generation of an explicit schedule of all operations, i.e., a completely unrolled program with as many statements as the total number of arithmetic operations to perform the SpGEMM computation. Instead, we take the following approach: 1) Use the results of the hypergraph partitioning to assign each row/columns of the input matrix A to one of the hypergraph partitions, based on maximum affinity, i.e., the partition containing the most arithmetic operations associated with that row/column; 2) Renumber the rows/columns of A based on the partition mapping, so that row/columns mapped to a partition are contiguously numbered in the new numbering; 3) Perform a symmetric row/column permutation of A based on the renumbering in the previous step and execute the SpGEMM code to perform the sparse matrix product of the reordered matrix.

Thus, we use the results of the hypergraph partitioning to reorder the original matrix, in the expectation that it could lead to better data locality and improved performance.

Alg. 4 shows pseudocode for the symmetric row/column reordering of a matrix based on hypergraph partitioning. After partitioning of hypergraph $\mathcal{H}(A, A)$, each vertex is assigned to a partition. Majority-voting is then performed for each row/column of the matrix to assign it to a partition.

For row/column u, this is done by tallying total counts for each hypergraph partition, of the number of multiply-add operations associated with row/column u (lines 3-7), and assigning it to the partition with the highest tally. The renumbering of row/column indexes is performed by contiguously numbering row/columns assigned to each partition, and ordering across the partitions in order of partition id. A permutation vector is formed by The number of row/columns assigned to each partition is first accumulated into vector ptr (lines 9-10), prefix-sum computed (lines 11-12), the permutation vector $reorder$ populated (lines 13-15), and the permuted matrix A_R generated (lines 16-17).

Similarly, matrix reordering was also performed for SpMV, based hypergraph on partitioning of the corresponding hypergraph.

Algorithm 4: Reorder Matrix For SpGEMM

input : Hypergraph $\mathcal{H}(A, A')$,// A' is deep copy of A
Num. Partitions $numPar$, SparseMatrix A[N][N]
output: SparseMatrix A_R[N][N]

1 par = Partition($\mathcal{H}(A, A)$,$numPar$)
2 **for** $1 \leq u \leq N$ **do**
3 **for** $(u, k) \in S_A \wedge (k, j) \in S_A$ **do**
4 vote[$par[v_{ukj}]$] = vote[$par[v_{ukj}]$] + 1
5 **for** $(k, u) \in S_A \wedge (i, k) \in S_A$ **do**
6 vote[$par[v_{iku}]$] = vote[$par[v_{iku}]$] + 1
7 $index[u]$ = s such that $\forall p$, vote[p] $\leq$ vote[s]
8 ptr[*] = 0
9 **for** $1 \leq u \leq N$ **do**
10 ptr[$index[u] + 1$] = ptr[$index[u] + 1$] + 1
11 **for** $1 < u \leq P$ **do**
12 ptr[u] = ptr[u] + ptr[u-1]
13 **for** $1 \leq u \leq N$ **do**
14 $reorder[u]$ = ptr[$index[u]$]
15 ptr[$index[u]$] = ptr[$index[u]$] +1
16 **for** $\forall (i, j) \in S_A$ **do**
17 $A_R[reorder[i]][reorder[j]]$ = A[i][j]

Fig. 6 presents experimental results comparing achieved performance and measured data volume to/from global memory, for both SpMV and SpGEMM. For most matrices, the reordering produces a slight improvement in data volume and performance, with significant changes in a few cases. It is interesting to note that the cases of significant improvement are usually not the same matrices for SpMV and SpGEMM. Overall, these experiments suggest that matrix reordering may potentially have a significant performance impact and exploration of alternate matrix reordering approaches is planned in ongoing/future work.

V. Experiments with Synthetic Banded Matrices

With all previous experiments, with or without matrix reordering, we can make the following observations:

- The actual measured data movement volume for SpGEMM is much higher than the upper bound determined via hypergraph partitioning.
- Even considering the actual measured data movement volume for SpGEMM, the achieved performance in GFLOPs is far from the roofline based on the achieved OI.

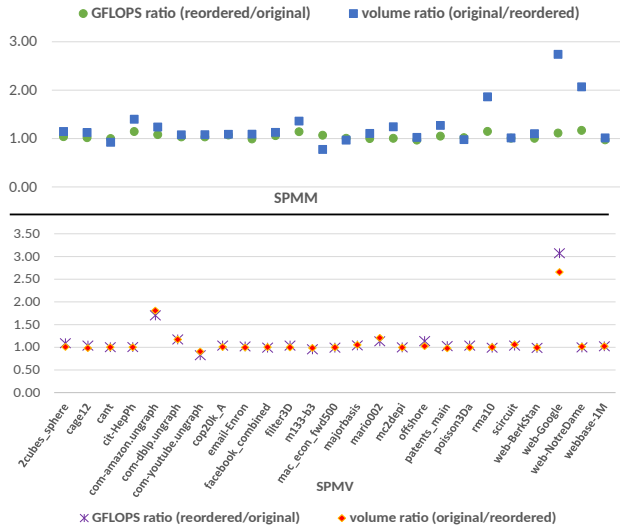

Figure 6. GFLOPS and data volume comparison for SPMM/SPMV on original/reordered matrices

The latter observation suggests that inadequate overall concurrency may be a significant factor in low SpGEMM performance. A challenge to gleaning insights into performance bottlenecks for SpGEMM is the fact that data elements from three sparse matrices are used in each elementary operation. Evan if two of the matrices are kept the same to perform C = A*A, different rows of A (with different sparsity patterns) are involved. In order to better control the variability and gain insights, we devised a set of experiments to perform SpGEMM on banded matrices, but represented using a CSR representation. Further, a random symmetric row/column permutation was performed for each tested banded matrix, and the randomized variant was also tested.

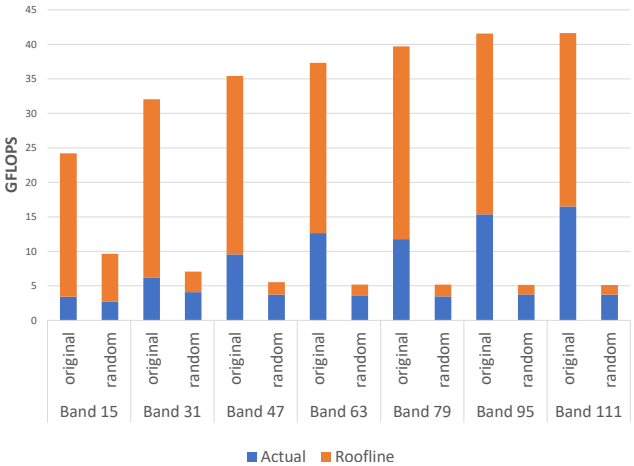

Figure 7. Banded Original vs Randomized

Fig. 7 presents the results of these experiments. The set of matrices had half-band sizes of 15, 31, 47, 63, 79, 95, and 111. For each matrix and its randomized variant, the stacked bar-chart shows actual achieved performance (blue) as well as the roofline performance bound based on measured data

volume to/from global memory. The main observations are as follows:

- The randomized variants achieve significantly lower performance than the contiguously represented banded matrices. This is a consequence of the significantly worse temporal locality in accessing rows of B for the randomized variant, as well as worse data coalescing for accesses to elements of C with the scatter-vector approach.
- As the band size increases (going from right to left in the chart), the ratio of roofline to actual performance decreases noticeably. This is more prominent for the randomized variants, which have a much higher data volume than the corresponding contiguous variants. This trend is suggestive that inadequate thread-level concurrency is a factor in the low performance of the SpGEMM implementation. As the band size increases, the average degree of concurrency with the scatter-vector approach increases, proportionally with the band size. At high band sizes, the execution is moving closer to a bandwidth-limited regime, as indicated by the lower ration of roofline performance bound to actual achieved performance.

The latter observation suggests the development of an adaptive work distribution scheme across virtual warps of a thread block, as developed in the next section.

VI. Enhancing Concurrency for SpGEMM

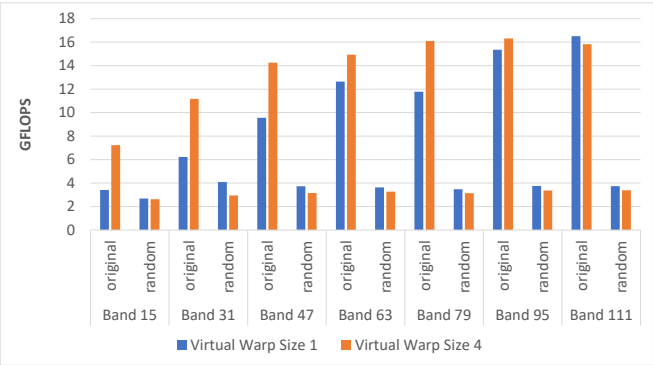

Figure 8. Performance variation wrt. virtual warp size

Scatter Vector Approach: Scatter vector is one approach to solve the index matching problem in SPGEMM. In Scatter vector approach, a vector of size n, where n is the number of columns in B is used to store the pointers to compacted elements in a row of resultant matrix "C". The rows of "A" matrix is processed sequentially. Before processing each row of A, the entire scatter vector is initialized to NULL value. For each "A" element in the current row, the corresponding "B" elements are identified and partial products are formed. For each such partial product, the corresponding column in the scatter vector is accessed. If the scatter vector contains a Non-NULL value, then the current partial contribution is added to the location pointed by the scatter vector. If the value is NULL, then a unique location is obtained from a memory pool and is initialized with the current partial product. The address of the obtained unique location is written to the corresponding column of the scatter vector. In the rest of this section, we describe our contributions to the scatter vector approach.

As shown in Figure 8, for many matrices our SpGEMM approach is far from the roofline limit which indicates that our approach is latency limited. Latency effects can be reduced by exposing more parallelism. The amount of available parallelism in GPUs is limited by the total number of warps that can be simultaneously active. "Achieved occupancy" metric from "Nvprof" indicate that our kernels achieve near optimal occupancy. This suggests that even with high occupancy, the effective parallelism is less. We observed that in Figure 8, the gap between achieved GFLOPS and theoretical GFLOPS is smaller for larger bands. Reasoning for the latter effect is as follows. In our approach, for each element in a row of "A" matrix, we assign all the threads in a thread block to process the corresponding "B" elements. The number of threads assigned to process each element in a given row of "A" depends on the total number of operations corresponding to that row (bin id). For example, bin 12 is assigned 128 threads. It may happen that even though the number of ops is high the number B elements is small. Since we process the elements in "A" matrix sequentially, if the number of elements in the corresponding rows in "B" matrix is less, then many threads will be idle. Note that even though the threads are idle, they have not exited the kernel. Hence, when measuring achieved occupancy, Nvprof considers these threads as active and reports a high occupancy. When compared to smaller bands, higher bands have higher number of non-zeros in "B". Hence, most of the threads are not idle/waiting which results in higher performance.

In order to improve effective concurrency, we implemented virtual warping. Each thread block processes multiple rows of "A" simultaneously and the entire threads in a thread block are equally divided and assigned to process each row of "A". For example, for a thread block of size 128, and virtual warp of size 4, each thread block will process 4 rows of "A" simultaneously. For each row of "A", 32 (128/4) threads are assigned to cyclically process the corresponding elements in "B". Virtual warping improved the performance of our approach.

Figure 8 compares the performance of our approach when virtual warp size is 1 and 4. Note that virtual warping is not beneficial for all bands sizes, which motivates an adaptive scheme. If the virtual warp size is greater than 4, then accesses to "B" elements may be partially uncoalesed. For eg. if the virtual warp size is 8, then 16 (128/8) threads will work on each row of A. The 32 elements of "B" (corresponding to two rows of "A") that are simultaneously required by a warp may not be contiguous which results in uncoalesed accesses. If the average number of non zeros in "B" is greater than 32, then the latter choice may not be beneficial. Consider another example when the average number of non-zeros in "B" is 128. In the latter case, a virtual warp size of 4 or 1 can achieve fully coalesed access; however the performance of virtual warp of size 1 can be better due to the following reason. When the virtual warp size is 1, each thread block is processing one row of "A" at a time. In contrast, when the virtual warp size is 4, four rows of "A" are processed at the same time, which increases

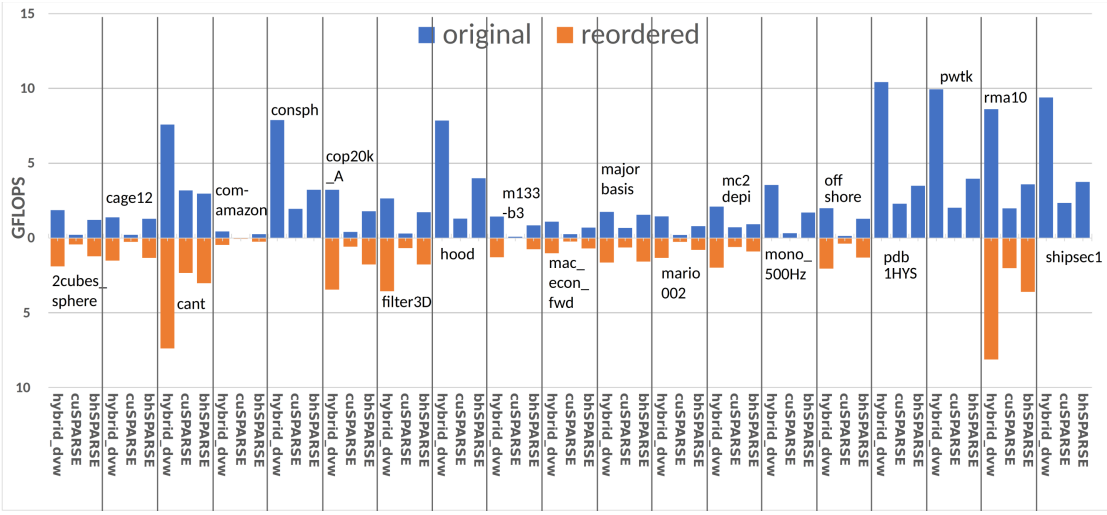

Figure 9. Performance Comparision between HybridSparse_dvw, Bhsparse and CUSparse

the cache pressure and reduces the cache hit rate.

Our adaptive virtual warping scheme, decides the virtual warp size depending on the average number of non-zeros in "B" for a given row of "A". For a given "C" row, the average number of non-zeros in "B" is determined by dividing the total ops by the number of elements in corresponding row of "A".

Fig. 9 presents performance data for the new SpGEMM implementation using the adaptive virtual warp scheme. Performance with two existing SpGEMM implementations, bhSPARSE and cuSPARSE are also presented. The data is presented as a mirrored set of bar charts, with the upright bars representing the original matrix, and the inverted bars representing performance after reordering using hypergraph partitioning (in a few cases, the PaToH hypergraph partitioner was unable to execute within the availablememory on our CPU system and no inverted bars are shown). It can be seen that the new HyprisSparse_dvw scheme consistently outperforms bhSPARSE and cuSPARSE on all the test matrices.

VII. Related Work

A. Bounds on Data Movement

Techniques have been developed to find schedule-independent lower bounds on the minimal amount of data movement between slow and fast memory [11] for computations abstracted as computational directed acyclic graphs (CDAGs). For a few regular algorithms, such as dense matrix-matrix and matrix-vector multiplication, FFT, and odd-even transposition sort, the lower-bounding technique based on a formalization using a red-blue pebble game [11] was used to derive tight parametric asymptotic lower bounds expressions for data movement complexity. For example, for dense matrix-matrix multiplication of $N \times N$ matrices on a processor with a cache capacity of C words, a tight lower bound for data movement between main-memory and cache was shown to be $O(N^3/\sqrt{C})$. Scaling constants for the lower bounds on matrix-matrix multiplication were provided by Irony et al. [12] and later improved by Dongarra et al. [8] to $1.72N^3/\sqrt{C}$. This lower bound is also generalizable for sparse matrix-matrix multiplication: $1.72 \times N_{Ops}/\sqrt{C}$, where N_{Ops} is the number of update (multiply-add) operations executed. However this bound is much lower than the simple data-footprint lower bound that we use, as discussed in Sec. III.

Ballard et al. [2] modeled data movement requirements for SpGEMM using a hypergraph, as we do in this paper. However they model data movement lower bounds for SpGEMM in terms of a minimal-partition hypergraph partitioning that satisfies some constraints, while we use a heuristic-based hypergraph partitioner (PaToH [3]) to determine *upper bounds* on data movement for SpGEMM, as elaborated in Sec. III.

B. SpMV and SpGEMM on GPUs

Over the last few years, there has been significant interest in developing efficient SpMV and SpGEMM implementations for GPUs, in part because they are key kernels for data analytics.

Nvidia's cuSPARSE library [18], [7] implements multiple SpMV implementations, for a number of sparse matrix formats, including CSR, COO (coordinate format), ELL (ELLPAck format), HYB (hybrid of ELL and COO). The multiplicity of formats is due to the fact that none of CSR, COO, ELL, HYB is uniformly superior in performance; the best format is dependent on the sparsity structure of the sparse matrix [19]. Recent developments have resulted in SpMV implementations that are quite consistently better than all variants in cuSPARSE, including CSR5 [15], Merge-CSR [17], and Hola [20]. In this paper, we use the CSR5 SpMV implementation for our experiments.

As discussed in Sec. II, a significant challenge with SpGEMM relative to SpMV is that of efficient load-balanced "index-matching" and accumulation of multiple additive contributions from products $A_{ik} * B_{kj}$ to form C_{ij}. Different approaches have been devised for this. The earliest efficient SpGEMM implementation was by Gustavson [10], who proposed the use of a scatter-vector for it. Our recent implementation of HybridSparse [13] is based on a combination of the scatter-vector approach and the ESC (Expand-Sort-Compress) approach. Nvidia's open-source CUSP library [5], [4] also uses the ESC approach. Nvidia's closed-source

library cuSPARSE uses hashing to address index-combining [18], [7]. A more recent GPU SpGEMM implementation by Anh rt al. [1] also uses hashing. Another approach to index-combining is "row-merging" (discussed in Sec. II), used by an implementation by Gremse et al. [9] and by Liu and Vinter for their bhSPARSE implementation [14], [16]. In this paper, we improve upon the HybridSparse implementation to create the HybridSparse-DVW variant with improved performance. Its performance is compared with cuSPARSE and bhSPARSE, whose implementations are publicly available. We did not compare with CUSP since previous studies have shown that it bhSPARSE achieves consistently higher performance [14], [16], [13].

A significant difference between the work presented in this paper from the above discussed efforts is that a primary motivation here has been to understand fundamental performance bottlenecks for SpGEMM, while previous efforts have had the primary goal of enhancing performance. A critical question of interest is to understand why SpGEMM performance is so far below roofline limits, in comparison to SpMV. While we were able to use insights from analysis to improve SpGEMM performance, the analysis indicates that there may be much further room for improvement, suggesting that continued research is warranted.

VIII. Conclusion

This paper has attempted to gain insights into the cause of relatively low performance of GPU implementations of general sparse matrix-matrix multiplication relative to sparse matrix-vector multiplication. A significant challenge with the former in comparison to the latter is the challenge of efficiently combining the various additive contributions to result data elements. All devised approaches to this problem cause additional data movement and/or introduce other sources of performance loss, such as hashing/sorting/merging operations, atomics, uncoalesced data access, thread divergence, etc.

In this paper, a systematic study was undertaken to identify any fundamental inherent data-movement bottlenecks for sparse matrix-matrix multiplication. A main conclusion are that the current low performance of SpGEMM implementations is not a consequence of any fundamental data movement requirements (as is the case for sparse matrix-vector multiplication). Inadequate operation-level concurrency was identified as a cause of performance loss for a scatter-vector based SpGEMM implementation and an improved implementation was devised that currently represents the highest performing GPU SpGEMM implementation available.

Acknowledgment

This work was supported in part by the Defense Advanced Research Projects Agency (DARPA) under Contract D16PC00183, and the National Science Foundation (NSF) through awards 1404995, 1440749, 1513120 and 1629548.

References

[1] P. N. Q. Anh, R. Fan, and Y. Wen. Balanced hashing and efficient gpu sparse general matrix-matrix multiplication. In *2016 International Conference on Supercomputing*, pages 36:1–36:12, New York, NY, USA, 2016. ACM.

[2] G. Ballard, A. Druinsky, N. Knight, and O. Schwartz. Hypergraph partitioning for sparse matrix-matrix multiplication. *ACM Transactions on Parallel Computing (TOPC)*, 3(3):18, 2016.

[3] Ü. V. Çatalyürek and C. Aykanat. Patoh (partitioning tool for hypergraphs)., 2011.

[4] S. Dalton and N. Bell. A C++ templated sparse matrix library, 2014.

[5] S. Dalton, L. Olson, and N. Bell. Optimizing sparse matrixmatrix multiplication for the gpu. *ACM Transactions on Mathematical Software (TOMS)*, 41(4):25, 2015.

[6] T. A. Davis and Y. Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1, 2011.

[7] J. Demouth. Sparse matrix-matrix multiplication on the gpu. In *Proceedings of the GPU Technology Conference*, 2012.

[8] J. Dongarra, J.-F. Pineau, Y. Robert, and F. Vivien. Matrix product on heterogeneous master-worker platforms. In *PPoPP*, pages 53–62, 2008.

[9] F. Gremse, A. Hofter, L. O. Schwen, F. Kiessling, and U. Naumann. Gpu-accelerated sparse matrix-matrix multiplication by iterative row merging. *SIAM Journal on Scientific Computing*, 37(1):C54–C71, 2015.

[10] F. G. Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Trans. Math. Softw.*, 4(3):250–269, Sept. 1978.

[11] J.-W. Hong and H. T. Kung. I/O complexity: The red-blue pebble game. In *STOC*, pages 326–333, 1981.

[12] D. Irony, S. Toledo, and A. Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *J. Parallel Distrib. Comput.*, 64(9):1017–1026, 2004.

[13] R. Kunchum, A. Chaudhry, A. Sukumaran-Rajam, Q. Niu, I. Nisa, and P. Sadayappan. On improving performance of sparse matrix-matrix multiplication on gpus. In *Proceedings of the International Conference on Supercomputing*, pages 14:1–14:11, New York, NY, USA, 2017. ACM.

[14] W. Liu and B. Vinter. An efficient gpu general sparse matrix-matrix multiplication for irregular data. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 370–381, 2014.

[15] W. Liu and B. Vinter. Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 339–350, New York, NY, USA, 2015. ACM.

[16] W. Liu and B. Vinter. A framework for general sparse matrix–matrix multiplication on gpus and heterogeneous processors. *Journal of Parallel and Distributed Computing*, 85:47–61, 2015.

[17] D. Merrill and M. Garland. Merge-based parallel sparse matrix-vector multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 58:1–58:12, Piscataway, NJ, USA, 2016. IEEE Press.

[18] NVIDIA. Nvidia cusparse library, 2017.

[19] N. Sedaghati, T. Mu, L.-N. Pouchet, S. Parthasarathy, and P. Sadayappan. Automatic selection of sparse matrix representation on gpus. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 99–108, New York, NY, USA, 2015. ACM.

[20] M. Steinberger, R. Zayer, and H.-P. Seidel. Globally homogeneous, locally adaptive sparse matrix-vector multiplication on the gpu. In *Proceedings of the International Conference on Supercomputing*, pages 13:1–13:11, New York, NY, USA, 2017. ACM.

[21] UFL. The suite sparse matrix collection, 2011.

[22] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.